scalasca

Scalasca 1.3 | User Guide

Scalable Automatic Performance Analysis

November 2010

The Scalasca Development Team
scalasca@fz-juelich.de

JÜLICH
FORSCHUNGSZENTRUM

German Research School
for Simulation Sciences

Copyright © 1998–2010 Forschungszentrum Jülich GmbH, Germany

Copyright © 2009–2010 German Research School for Simulation Sciences GmbH, Germany

Copyright © 2003–2008 University of Tennessee, Knoxville, USA

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the names of Forschungszentrum Jülich GmbH, the German Research School for Simulation Sciences GmbH, or the University of Tennessee, Knoxville, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents

1 Introduction **1**
1.1 How to read this document . . . 1
1.2 Performance optimization cycle . . . 2
1.3 Scalasca overview . . . 3

2 Getting started **7**
2.1 Instrumentation . . . 8
2.2 Runtime measurement collection & analysis . . . 9
2.3 Analysis report examination . . . 10
2.4 A full workflow example . . . 12

3 Application instrumentation **19**
3.1 Automatic compiler instrumentation . . . 21
3.2 Manual region instrumentation . . . 22
3.3 Semi-automatic instrumentation . . . 24
3.4 Automatic source-code instrumentation using PDT . . . 25
3.5 Selective instrumentation . . . 27

4 Measurement collection & analysis **29**
4.1 Nexus configuration . . . 29
4.2 Measurement configuration . . . 31
4.3 Measurement and analysis of hardware counter metrics . . . 33
4.4 Automatic parallel event trace analysis . . . 34
4.5 Automatic sequential event trace analysis . . . 36

5 Additional utilities **39**
5.1 Additional EPILOG event trace utilities . . . 39
5.2 Trace converters . . . 39
5.3 Recording user-specified virtual topologies . . . 40

A MPI wrapper affiliation **43**
A.1 Function to group . . . 43
A.2 Group to function . . . 52

Bibliography **63**

1 Introduction

Supercomputing is a key technology of modern science and engineering, indispensable to solve critical problems of high complexity. As a prerequisite for the productive use of today's large-scale computing systems, the HPC community needs powerful and robust performance analysis tools that make the optimization of parallel applications both more effective and more efficient.

Developed at the Jülich Supercomputing Centre, Scalasca is a performance analysis toolset that has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XT, but also suitable for smaller HPC platforms using MPI and/or OpenMP. Scalasca supports an incremental performance analysis process that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. A distinctive feature of Scalasca is the ability to identify wait states that occur, for example, as a result of unevenly distributed workloads. Especially when trying to scale communication intensive applications to large processor counts, such wait states can present severe challenges to achieving good performance. Compared to its predecessor KOJAK [12], Scalasca can detect such wait states even in very large configurations of processes using a novel parallel trace-analysis scheme [5].

1.1 How to read this document

This user guide is structured into three parts:

- This introductory chapter gives a short introduction into performance analysis in general and the components of the Scalasca toolset in particular. If you are already familiar with performance analysis of parallel scientific applications you might skip the following section and continue reading directly with Section 1.3.

- The next part in Chapter 2 introduces the basic steps and commands required for initial performance analyses of parallel applications. It also includes a full example describing the Scalasca performance analysis workflow.

- The remainder of this user guide in Chapters 3 to 5 provide a more in-depth discussion of the individual steps in evaluating the performance of a parallel application.

1.2 Performance optimization cycle

Regardless of whether an application should be optimized for single-core performance or for scalability, the basic approach is very similar. First, the behavior of the application needs to be monitored, and afterwards the recorded behavior can be evaluated to draw conclusions for further improvement. This is an iterative process that can be described by a cycle, the so-called performance optimization cycle. When broken down into phases, it is comprised of:

- Instrumentation
- Measurement
- Analysis
- Presentation
- Evaluation
- Optimization of the code

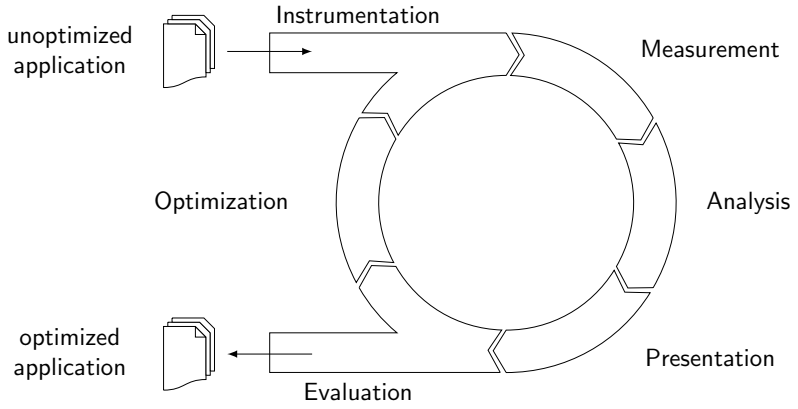

Figure 1.1: Performance optimization cycle

As shown in Figure 1.1, the user starts with the original (i.e., unoptimized) application, which enters the optimization cycle in the instrumentation phase. Instrumentation describes the process of modifying the application code to enable measurement of performance relevant data during the application run. In the context of Scalasca, this can be achieved by different mechanisms, such as source-code instrumentation, automatic compiler-based instrumentation or linking with pre-instrumented libraries. Instrumentation on the source-code level can be done by introducing additional instructions into the source code prior to compilation. On most systems, this process can be automated by using special features of the compiler. However, this approach typically does not allow a fine-grained control of the instrumention. The third method is to use pre-instrumented libraries, which contain instrumented versions of the relevant library functions. The

Message-Passing Interface standard MPI [8] provides a special interface for this kind of instrumentation, the so-called PMPI interface. As this interface is defined in the MPI standard, its API is portable and creates an opportunity for tool developers to provide a single portable measurement library for multiple different MPI implementations.

When the instrumented code is executed during the measurement phase, performance data is collected. This can be stored as a profile or an event trace, depending on the desired level of information needed. The additional instructions inserted during instrumentation and associated measurement storage require resources (memory as well as CPU time). Therefore the application execution is affected to a certain degree. Perturbation by the additional measurement instructions may be small enough to get a fairly accurate view of the application behavior. However, certain application properties like frequently executed regions with extremely small temporal extent, will always lead to a high perturbation. Thus the measurement of those regions must be avoided.

The measurement data can then be analyzed after application execution. If a detailed event trace has been collected, more sophisticated dependencies between events occuring on different processes can be investigated, resulting in a more detailed analysis report. Especially inter-process event correlations can usually only be analyzed by a post-mortem analysis. The information which is needed to analyze these correlations are usually distributed over the processes. Transferring the data during normal application runtime would lead to a significant perturbation during measurement, as it would require application resources on the network for this.

After analyzing the collected data, the result needs to be presented in an analysis report. This leads to the next phase in the performance optimization cycle, namely the presentation phase. At this stage, it is important to reduce the complexity of the collected performance data to ease evaluation by the user. If the presented data is too abstract, performance critical event patterns might not be recognized by the user. If it is too detailed, the user might drown in too much data. User guidance is therefore the key to productive application optimization.

In the evaluation phase, conclusions are drawn from the presented information, leading to optimization hypotheses. The user proposes optimization strategies for the application, which are then implemented in the following optimization phase. Afterwards, the effectiveness of the optimization has to be verified by another pass through the performance optimization cycle. When the user is satisfied with the application performance during evaluation, and no further optimization is needed, the instrumentation can be disabled, and the performance of the uninstrumented application execution can be assessed.

1.3 Scalasca overview

Scalasca supports measurement and analysis of the MPI, OpenMP and hybrid MPI/OpenMP programming constructs most widely used in highly scalable HPC applications written in C, C++ and Fortran on a wide range of current HPC platforms.

Usage is primarily via the `scalasca` command with appropriate action flags. Figure 1.2 shows the basic analysis workflow supported by Scalasca. Before any performance data can be collected, the target application needs to be instrumented. Instrumentation means, that the code must be modified to record performance-relevant events whenever they occur. On most systems, this can be done completely automatically using compiler support. On other systems, a mix of manual and automatic instrumentation mechanisms is offered. When executing the instrumented code on a parallel machine, the user can generate a summary report (also known as *profile*) with aggregate performance metrics for individual function call paths. Furthermore, event traces can be generated by recording individual runtime events from which a profile or a time-line visualization can later be produced. The runtime summarization capability is useful to obtain an overview of the performance behavior and also to optimize the instrumentation for later trace generation. Since traces tend to become very large, and inappropriate instrumentation and measurement configuration will compromise the resulting analysis, this step is highly recommended.

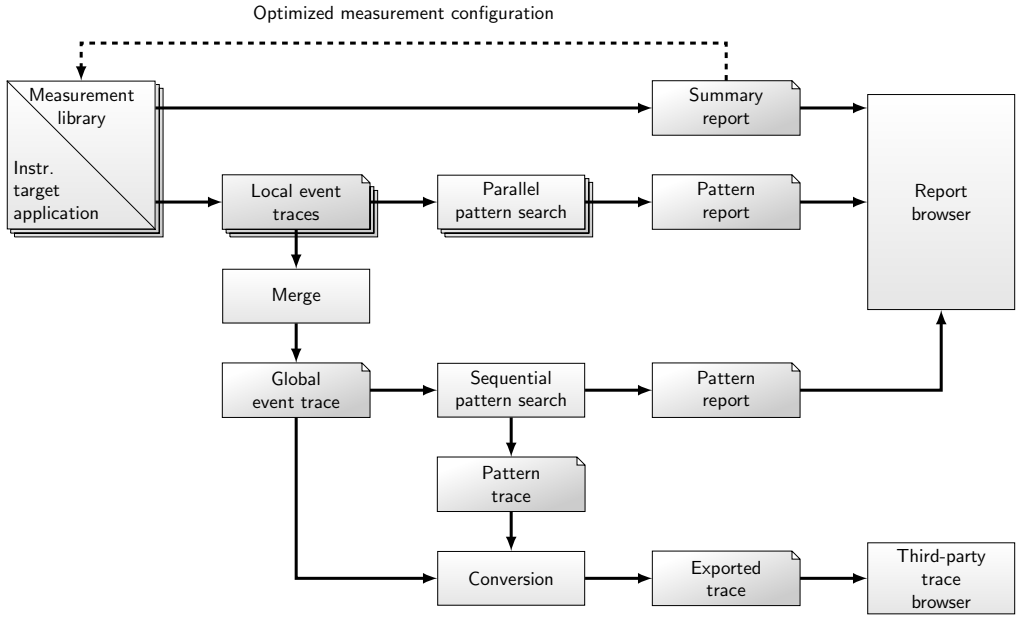

Figure 1.2: Scalasca's performance analysis workflow

When tracing is enabled, each process generates a trace file containing records for all its process-local events. After program termination, Scalasca reloads the trace files back into main memory and analyzes them in parallel using as many CPUs as have been used for the target application itself. During the analysis, Scalasca searches for characteristic patterns indicating wait states and related performance properties, classifies detected instances by category and quantifies their significance. The result is a pattern-analysis report similar in structure to the summary report but enriched with higher-level communication and synchronization inefficiency metrics. Both summary and pattern reports

contain performance metrics for every function call path and system resource which can be interactively explored in a graphical report explorer (see Figure 2.1 for an example). As an alternative to the automatic analysis, the event traces can be converted and investigated using third-party trace browsers such as Paraver [7, 3] or Vampir [9, 6], taking advantage of their powerful time-line visualizations and rich statistical functionality.

2 Getting started

This chapter will provide a hands-on introduction to the use of the Scalasca toolset on the basis of the analysis of an example application. The most prominent features will be addressed, and at times a reference to later chapters with more in-depth information on the corresponding topic will be given.

Use of Scalasca involves three phases: program instrumentation, execution measurement and analysis, and analysis report examination. The `scalasca` command provides action options that invoke the corresponding commands `skin`, `scan` and `square`.

These actions are:

1. `scalasca -instrument`

   is used to insert calls to the Scalasca measurement system into the application's code, either automatically, semi-automatically or by linking with pre-instrumented libraries.

2. `scalasca -analyze`

   is used to control the measurement environment during the application execution, and to automatically perform trace analysis after measurement completion if tracing was enabled.

   The Scalasca measurement system supports runtime summarization and/or event trace collection and analyses, optionally including hardware-counter information.

3. `scalasca -examine`

   is used to postprocess the analysis report generated by the measurement runtime summarization and/or post-mortem trace analysis, and to start Scalasca's analysis report examination browser CUBE3.

To get a brief usage summary, call the `scalasca` command with no arguments, or use `scalasca -h` to open the Scalasca Quick Reference (if a suitable PDF viewer can be found).

The following three sections provide a quick overview of each of these actions and how to use them during the corresponding step of the performance analysis, before a full workflow example is presented in Section 2.4.

2.1 Instrumentation

To make measurements with the Scalasca measurement system, user application programs need to be instrumented, i.e., at specific important points (events) during the application run, special measurement calls have to be inserted. In addition to an almost automatic approach using compiler-inserted instrumentation (Section 3.1), semi-automatic "POMP" (Section 3.3) and manual instrumentation (Section 3.2) approaches are also supported. In addition, automatic source-code instrumentation by the PDT instrumentor (Section 3.4) can be used if Scalasca is configured accordingly.

For pure OpenMP or hybrid MPI/OpenMP applications, or when using the semi-automatic "POMP" directive-based approach, the OPARI source-code instrumenter is used internally. Read the OPARI section in the OPEN_ISSUES file provided as part of the Scalasca documentation to be aware of its current limitations and how to work around some of them, including:

- parallel builds (e.g., using `gmake -j`)
- multiple applications built within a single directory
- applications with sources spread over multiple directories

All the necessary instrumentation of user, OpenMP and MPI functions is handled by the Scalasca instrumenter, which is accessed through the `scalasca -instrument` command. Therefore, the compile and link commands to build the application that is to be analyzed should be prefixed with `scalasca -instrument` (e.g., in a Makefile).

For example, to instrument the application executable `myprog` generated from the two source files `myprog1.f90` and `myprog2.f90`, replace the combined compile and link command

```
mpif90 myprog1.f90 myprog2.f90 -o myprog
```

by the following command using the Scalasca instrumenter:

```
scalasca -instrument mpif90 myprog1.f90 myprog2.f90 -o myprog
```

Note:

The instrumenter must be used with the link command. However, not all object files need to be instrumented, and it is often sufficient to only instrument source modules containing OpenMP and/or MPI code.

When using Makefiles, it is often convenient to define a "preparation preposition" placeholder (e.g., `PREP`) which can be prefixed to (selected) compile and link commands:

```
MPICC  = $(PREP) mpicc
MPICXX = $(PREP) mpicxx
MPIF90 = $(PREP) mpif90
```

These can make it easier to prepare an instrumented version of the program by building with

```
make PREP="scalasca -instrument"
```

while default builds (without specifying `PREP` on the command line) remain fully optimized and without instrumentation.

Although generally most convenient, automatic function instrumentation may result in too many and/or too disruptive measurements, which can be addressed with selective instrumentation and measurement filtering (see Sections 3.4 and 3.5).

2.2 Runtime measurement collection & analysis

The Scalasca measurement collection & analysis nexus accessed through the `scalasca -analyze` command integrates the following steps:

- measurement configuration
- application execution
- collection of measured data
- automatic post-mortem trace analysis (if configured)

To make a performance measurement using an instrumented executable, the target application execution command is prefixed with the `scalasca -analyze` command:

```
scalasca -analyze [options] \
         $MPIEXEC $MPI_FLAGS <target> [target args]
```

For non-MPI (i.e., serial and pure OpenMP) applications, the MPI launch command and associated flags should be omitted.

A unique directory is used for each measurement experiment, which must not already exist when measurement starts. Measurement is aborted if the specified directory exists. A default name for each EPIK measurement archive directory is created from the name of the target application executable, the run configuration (e.g., number of processes and `OMP_NUM_THREADS` specified), and the measurement configuration. This archive name has an `epik_` prefix and its location can be explicitly specified to Scalasca with the `-e <path>` option or changed via configuration variables.

When the measurement has completed, the measurement archive directory contains various log files and one or more analysis reports. By default, runtime summarization is used to provide a summary report of the number of visits and time spent on each callpath by each process. For MPI measurements, MPI time and message and file I/O statistics are included. For OpenMP measurements, OpenMP-specific metrics are calculated. Hybrid

OpenMP/MPI measurements contain both sets of metrics. If hardware counter metrics were requested, these are also included in the summary report.

Event trace data can also be collected as a part of the measurement, producing an EPILOG trace file for each process. To collect event trace data as part of the measurement, use the `scalasca -analyze -t` command (or alternatively set the configuration variable `EPK_TRACE=1`). In this case, experiment trace analysis is automatically initiated after measurement is complete. Note that for pure OpenMP codes, the automatic trace analysis does not yet provide a more detailed wait-state analysis compared to runtime summarization. However, you might still want to collect traces for visualization in a graphical trace browser.

The `scalasca -analyze -n` preview mode can be used to show (but not actually execute) the measurement and analysis launch commands, along with various checks to determine the possible success. Additional informational commentary (via `-v`) may also be revealing, especially if measurement or analysis was unsuccessful.

In case of problems which are not obvious from reported errors or warnings, set the configuration variable `EPK_VERBOSE=1` before executing the instrumented application to see control messages of the Scalasca measurement system. This might help to track down the problem or allow a detailed problem report to be given to the Scalasca developers. (Since the amount of messages may be overwhelming, use an execution configuration that is as small and short as possible.)

When using environment variables in a cluster environment, make sure that they have the same value for all application processes on *all* nodes of the cluster. Some cluster environments do not automatically transfer the environment when executing parts of the job on remote nodes of the cluster, and may need to be explicitly set and exported in batch job submission scripts.

2.3 Analysis report examination

The results of the automatic analysis are stored in one or more reports in the experiment archive. These reports can be processed and examined using the `scalasca -examine` command on the experiment archive:

```
scalasca -examine epik_<title>
```

Post-processing is done the first time that an archive is examined, before launching the CUBE3 report viewer. If the `scalasca -examine` command is executed on an already processed experiment archive, or with a CUBE file specified as argument, the viewer is launched immediately.

A short textual score report can be obtained without launching the viewer:

```
scalasca -examine -s epik_<title>
```

This score report comes from the `cube3_score` utility and provides a breakdown of the different types of region included in the measurement and their estimated associated trace buffer capacity requirements, aggregate trace size (`total_tbc`) and largest process trace size (`max_tbc`), which can be used to specify an appropriate `ELG_BUFFER_SIZE` for a subsequent trace measurement.

The CUBE3 viewer can also be used on an experiment archive or CUBE file:

```
cube3 epik_<title>
cube3 <file>.cube
```

However, keep in mind that no post-processing is performed in this case, so that only a subset of Scalasca analyses and metrics may be shown.

2.3.1 Using CUBE3

The following paragraphs provide a very brief introduction of the CUBE3 usage. To effectively use the GUI, you should also read the CUBE3 manual provided with the Scalasca distribution.

CUBE3 is a generic user interface for presenting and browsing performance and debugging information from parallel applications. The underlying data model is independent from particular performance properties to be displayed. The CUBE3 main window consists of three panels containing tree displays or alternate graphical views of analysis reports. The left panel shows performance properties of the execution, the middle pane shows the call-tree or a flat profile of the application, and the right tree either shows the system hierarchy consisting of machines, compute nodes, processes, and threads or a topological view of the application's processes and threads. All tree nodes are labeled with a metric value and a colored box which can help identify hotspots. The metric value color is determined from the proportion of the total (root) value or some other specified reference value.

A click on a performance property or a call path selects the corresponding node. This has the effect that the metric value held by this node (such as execution time) will be further broken down into its constituents. That is, after selecting a performance property, the middle panel shows its distribution across the call tree. After selecting a call path (i.e., a node in the call tree), the system tree shows the distribution of the performance property in that call path across the system locations. A click on the icon left to a node in each tree expands or collapses that node. By expanding or collapsing nodes in each of the three trees, the analysis results can be viewed on different levels of granularity.

To obtain the exact definition of a performance property, select "Online Description" in the context menu associated with each performance property, which is accessible using the right mouse button. A brief description can be obtained from the menu option "Info". Further information is also available at the Scalasca website

http://www.scalasca.org/

CUBE3 also provides a number of algebra utilities which are command-line tools that operate on analysis reports. (The utililties currently only work on the CUBE files within experiment archive directories, not on the archives themselves.) Multiple analysis reports can be averaged with `cube3_mean` or merged with `cube3_merge`. The difference between two analysis reports can be calculated using `cube3_diff`. Finally, a new analysis report can be generated after pruning specified call trees and/or specifying a call-tree node as a new root with `cube3_cut`. The latter can be particularly useful for eliminating uninteresting phases (e.g., initialization) and focussing the analysis on a selected part of the execution. Each of these utilities generates a new CUBE-formated report as output.

The `cube3_score` utility can be used to estimate trace buffer requirements from summary or trace analysis reports. If sufficient memory is physically available, this can be specified in the ELG_BUFFER_SIZE configuration variable for a subsequent trace collection. Detailed region output (`cube3_score -r`) can also be examined to identify frequently executed regions that may adversely impact measurement and not be considered valuable as part of the analysis. Such regions without OpenMP and MPI operations may be appropriate for exclusion from subsequent experiments via selective instrumentation and measurement (see Sections 3.4 and 3.5). Trace buffer capacity can be saved by eliminating certain functions from the measurement. This could be done by providing a filter file, which lists the function names of the functions to be excluded. A potential filter file can be evaluated with the option `-f <filter_file>`.

2.4 A full workflow example

The previous sections introduced the general usage of Scalasca. This section will guide through an example analysis of a solver kernel called SOR, solving the Poisson equation using a red-black successive over-relaxation method. The environment used in the following examples is the IBM Blue Gene/P, present at the Jülich Supercomputing Centre of Forschungszentrum Jülich. The commands and outputs presented in this section might differ from the commands and outputs of your system.

By default, Scalasca uses the automatic compiler-based instrumentation feature. This is usually the best first approach, when you don't have detailed knowledge about the application and need to identify the hotspots in your code. SOR consists of only a single source file, which can be compiled and linked using the following two commands:

```
scalasca -instrument mpixlc -c sor.c
scalasca -instrument mpixlc sor.o -o sor.x
```

Now the instrumented binary must be executed on the system. On supercomputing systems, users usually have to submit their jobs to a batch system and are not allowed to start

parallel jobs directly. Therefore, the call to the `scalasca` command has to be provided within a batch script, which will be scheduled for execution when the required resources are available.

The syntax of the batch script differs between the different scheduling systems. However, common to every batch script format is a passage where all shell commands can be placed that will be executed. Here, the call to the Scalasca analyzer has to be placed in front of the application execution command:

```
scalasca -analyze mpirun -mode vn -np 128 ./sor.x
```

Ensure that the `scalasca` command is accessible when the batch script is executed, e.g., by updating the PATH if necessary. The parameters `-mode` and `-np` are options of the `mpirun` command on the Blue Gene system and other launchers will have different flags and syntax.

The Scalasca analyzer will take care of certain control variables, which assist in the measurement of your application. The default behavior of the Scalasca analyzer is to create a summary file, and not to create a detailed event trace, as indicated by the initial messages from the EPIK measurement system.

```
S=C=A=N: Scalasca 1.3 runtime summarization
S=C=A=N: ./epik_sor_vn128_sum experiment archive
S=C=A=N: Collect start
mpirun -mode vn -np 128 ./sor.x
[00000]EPIK: Created new measurement archive ./epik_sor_vn128_sum
[00000]EPIK: Activated ./epik_sor_vn128_sum [NO TRACE]

            [... Application output ...]

[00000]EPIK: Closing experiment ./epik_sor_vn128_sum
...
[00000]EPIK: Closed experiment ./epik_sor_vn128_sum
S=C=A=N: Collect done
S=C=A=N: ./epik_sor_vn128_sum complete.
```

After successful execution of the job, a summary report file is created within a new measurement directory. In this example, the automatically generated name of the measurement directory is `epik_sor_vn128_sum`, indicating that the job was executed in Blue Gene's virtual node mode (`-mode vn`) with 128 processes (`-np 128`). The suffix `_sum` refers to a runtime summarization experiment. The summary analysis report can then be post-processed and examined with the Scalasca report browser:

```
scalasca -examine epik_sor_vn128_sum
INFO: Post-processing runtime summarization report ...
INFO: Displaying ./epik_sor_vn128_sum/summary.cube ...
```

Figure 2.1 shows a screenshot of the Scalasca report browser CUBE3 with the summary analysis report of SOR opened. The examination of the application performance summary may indicate several influences of the measurement on your application behavior. For example, frequently executed, short functions may lead to significant pertubation and would be prohibitive to trace: these need to be eliminated before further investigations using trace analysis are taken into account.

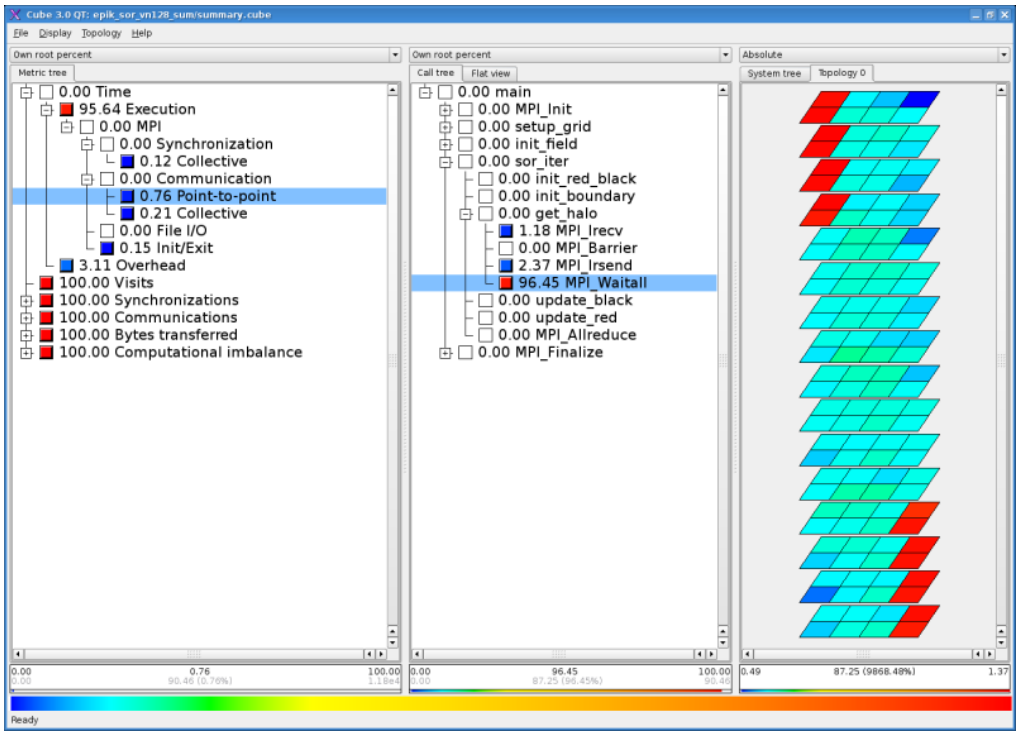

Figure 2.1: Viewing a runtime summary in CUBE

During trace collection, information about the application's execution behavior is recorded in so-called event streams. The number of events in the streams determines the size of the buffer required to hold the stream in memory. To minimize the amount of memory required, and to reduce the time to flush the event buffers to disk, only the most relevant function calls should be instrumented.

When the complete event stream is larger than the memory buffer, it has to be flushed to disk during application runtime. This flush impacts application performance, as flushing is not coordinated between processes, and runtime imbalances are induced into the measurement. The Scalasca measurement system uses a default value of 10 MB per process or thread for the event trace: when this is not adequate it can be adjusted to minimize or eliminate flushing of the internal buffers. However, if too large a value is specified for the buffers, the application may be left with insufficient memory to run, or run adversely with paging to disk. Larger traces also require more disk space (at least temporarily, un-

til analysis is complete), and are correspondingly slower to write to and read back from disk. Often it is more appropriate to reduce the size of the trace (e.g., by specifying a shorter execution, or more selective instrumentation and measurement), than to increase the buffer size.

To estimate the buffer requirements for a trace measurement, `scalasca -examine -s` will generate a brief overview of the estimated maximal number of bytes required.

```
scalasca -examine -s epik_sor_vn128_sum

[cube3_score epik_sor_vn128_sum/summary.cube]
Reading ./epik_sor_vn128_sum/summary.cube... done.
Estimated aggregate size of event trace (total_tbc): 25698304 bytes
Estimated size of largest process trace (max_tbc):   215168 bytes
(When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate flushes
 or reduce requirements using a file listing USR regions to be filtered.)

flt  type       max_tbc         time       % region
      ANY        215168     11849.04  100.00 (summary) ALL
      MPI        195728       147.47    1.24 (summary) MPI
      COM          9696       465.93    3.93 (summary) COM
      USR          9744     11235.64   94.82 (summary) USR
```

The line at the top of the table referring to `ALL` provides the aggregate information for all executed functions. In this table, the column `max_tbc` refers to the maximum of the trace buffer capacity requirements determined for each process in bytes. If `max_tbc` exceeds the buffer size available for the event stream in memory, intermediate flushes during measurement will occur. To prevent flushing, you can either increase the event buffer size or exclude a given set of functions from measurement.

To aid in setting up an appropriate filter file, this "scoring" functionality also provides a breakdown by different categories, determined for each region according to its type of call path. Type `MPI` refers to function calls to the MPI library and type `OMP` either to OpenMP regions or calls to the OpenMP API. User-program routines on paths that directly or indirectly call MPI or OpenMP provide valuable context for understanding the communication and synchronization behaviour of the parallel execution, and are distinguished with the `COM` type from other routines that are involved with purely local computation marked `USR`.

Routines with type `USR` are typically good candidates for filtering, which will effectively make them invisible to measurement and analysis (as if they were "inlined"). `COM` routines can also be filtered, however, this is generally undesirable since it eliminates context information. Since `MPI` and `OMP` regions are required by Scalasca analyses, these cannot be filtered.

By comparing the trace buffer requirements with the time spent in the routines of a particular group, the initial scoring report will already indicate what benefits can be expected from filtering. However, to actually set up the filter, a more detailed examination

is required. This can be achieved by applying the `cube3_score` utility directly on the post-processed summary report using the additional command-line option `-r`:

```
cube3_score -r epik_sor_vn128_sum/summary.cube

Reading summary.cube... done.
Estimated aggregate size of event trace (total_tbc): 25698304 bytes
Estimated size of largest process trace (max_tbc):   215168 bytes
(When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate flushes
 or reduce requirements using a file listing USR regions to be filtered.)

flt  type       max_tbc          time       % region
      ANY        215168      11849.04  100.00 (summary) ALL
      MPI        195728        147.47    1.24 (summary) MPI
      COM          9696        465.93    3.93 (summary) COM
      USR          9744      11235.64   94.82 (summary) USR

      MPI         80000          2.14    0.02 MPI_Irsend
      MPI         73600          1.07    0.01 MPI_Irecv
      MPI         16040         20.77    0.18 MPI_Allreduce
      MPI         16000         14.32    0.12 MPI_Barrier
      MPI          9600         87.25    0.74 MPI_Waitall
      COM          9600        304.28    2.57 get_halo
      USR          4800       5432.60   45.85 update_red
      USR          4800       5432.87   45.85 update_black
      MPI           240          0.54    0.00 MPI_Gather
      MPI           200          3.63    0.03 MPI_Bcast
      USR            48        368.66    3.11 TRACING
      USR            48          0.50    0.00 looplimits
      MPI            24          0.52    0.00 MPI_Finalize
      USR            24          0.54    0.00 init_boundary
      USR            24          0.48    0.00 init_red_black
      COM            24          2.88    0.02 sor_iter
      COM            24        156.25    1.32 init_field
      COM            24          0.82    0.01 setup_grid
      MPI            24         17.23    0.15 MPI_Init
      COM            24          1.70    0.01 main
```

(The basic form of this command is reported when running `scalasca -examine -s`.) As the maximum trace buffer required on a single process for the SOR example is approx. 215 KB, there is no need for filtering in this case.

Note:

Filtering will not prevent the function from being instrumented. Hence, measurement overhead can currently not be completely eliminated on filtered functions when automatic compiler-based instrumentation is used.
Once the configuration of buffer sizes and/or filters have been determined, make sure they are specified for subsequent (tracing) measurements, via environment variables or an EPIK.CONF measurement configuration file in the working directory. A filter

file can also be specified with `-f filter_file` to measurements with `scalasca -analyze`.
Before initiating a trace measurement experiment, ensure that the filesystem where the experiment will be created is appropriate for parallel I/O (typically /scratch or /work rather than /home) and that there will be sufficient capacity (and/or quota) for the expected trace.

When all options of the Scalasca measurement system are set in a way that measurement overhead and space requirements are minimized, a new run of the instrumented application can be performed, passing the `-t` option to `scalasca -analyze`. This will enable the tracing mode of the Scalasca measurement system. Additionally, the parallel post-mortem trace analyzer searching for patterns of inefficient communication behavior is automatically started after application completion.

```
scalasca -analyze -t mpirun -mode vn -np 128 ./sor.x

S=C=A=N: Scalasca 1.3 trace collection and analysis
S=C=A=N: ./epik_sor_vn128_trace experiment archive
S=C=A=N: Collect start
mpirun -mode vn -np 128 ./sor.x
[00000]EPIK: Created new measurement archive ./epik_sor_vn128_trace
[00000]EPIK: Activated ./epik_sor_vn128_trace [10000000 bytes]

            [... Application output ...]

[00000]EPIK: Closing experiment ./epik_sor_vn128_trace
[00000]EPIK: Flushed file ./epik_sor_vn128_trace/ELG/00000
...
[00013]EPIK: Flushed file ./epik_sor_vn128_trace/ELG/00013
[00000]EPIK: Closed experiment ./epik_sor_vn128_trace
S=C=A=N: Collect done
S=C=A=N: Analysis start
mpirun -mode vn -np 128 scout.mpi ./epik_sor_vn128_trace
            [... SCOUT output ...]
S=C=A=N: Analysis done
S=C=A=N: ./epik_sor_vn128_trace complete.
```

This creates an experiment archive directory `epik_sor_vn128_trace`, distinguishing it from the previous summary experiment through the suffix `_trace`. A separate trace file per MPI rank is written directly into a subdirectory when measurement is closed, and the parallel trace analyzer SCOUT is automatically launched to analyze these trace files and produce an analysis report. This analysis report can then be examined using the same commands and tools as the summary experiment.

```
scalasca -examine epik_sor_vn128_trace
INFO: Post-processing trace analysis report ...
INFO: Displaying ./epik_sor_vn128_trace/trace.cube ...
```

The screenshot in Figure 2.2 shows that the trace analysis result at first glance provides the same information as the summary result. However, the trace analysis report is enriched with additional performance metrics which show up as sub-metrics of the summary properties, such as the fraction of Point-to-point Communication Time potentially wasted due to Late Sender situations where early receives had to wait for sends to be initiated. That is, the trace analysis can reveal detail of inefficient execution behavior.

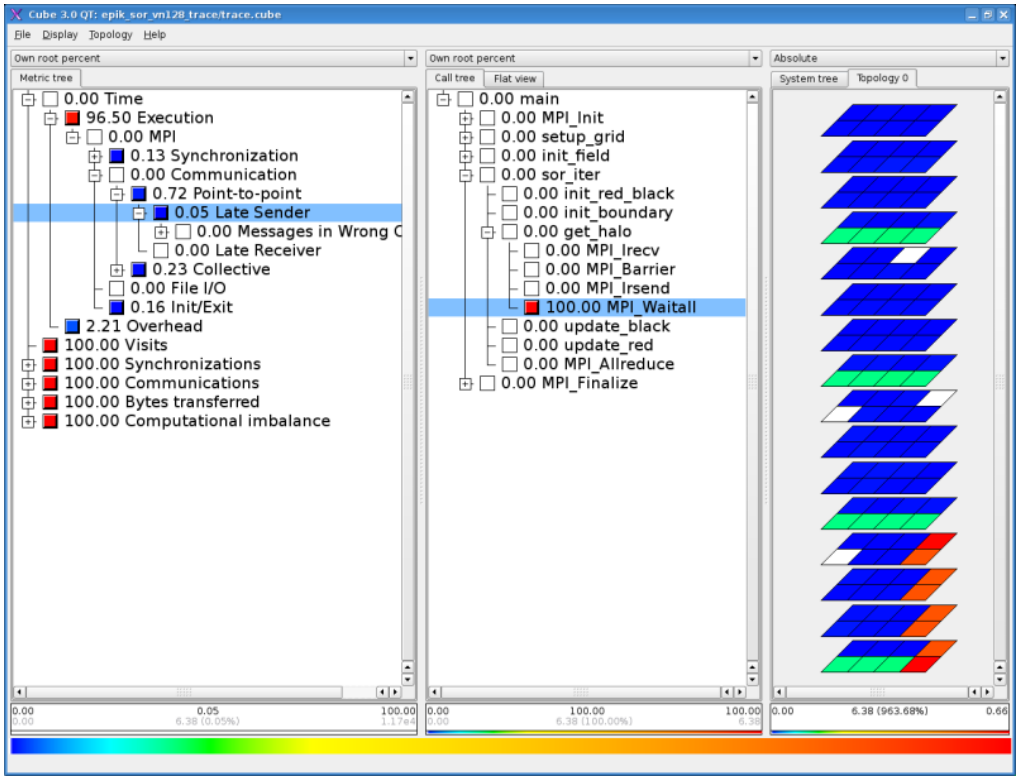

Figure 2.2: Determine a Late Sender in CUBE3.

The filesystem requirements for an EPILOG event trace and its analysis are much higher than for a runtime summary. The runtime of a batch job will also increase due to additional file I/O at the end of measurement and for analysis. After successful tracing, the Scalasca measurement has created a directory containing the event trace and its analysis files. The default behavior of Scalasca in tracing mode is to create a runtime summary report (stored in `summary.cube`) as well as a trace analysis report (stored in `trace.cube`).

After successful trace analysis, and before moving the experiment archive, the trace files can be removed by deleting the `ELG` subdirectory in the experiment archive.

3 Application instrumentation

Scalasca provides several possibilities to instrument user application code. Besides the automatic compiler-based instrumentation (Section 3.1), it provides manual instrumentation using the EPIK API (Section 3.2), semi-automatic instrumentation using POMP directives (Section 3.3) and, if configured, automatic source-code instrumentation using the PDToolkit-based instrumentor (Section 3.4). Additionally, Scalasca provides a filtering capability for excluding instrumented user routines from measurement (Section 3.5) if automatic compiler-based instrumentation is used.

As well as user routines and specified source regions, Scalasca currently supports the following kinds of events:

- **MPI library calls**:

  Instrumentation is accomplished using the standard MPI profiling interface PMPI. To enable it, the application program has to be linked against the EPIK MPI (or hybrid) measurement library plus MPI-specific libraries. Note that the EPIK libraries must be linked *before* the MPI library to ensure interposition will be effective.

- **OpenMP directives & API calls**:

  The Scalasca measurement system uses the OPARI tool for instrumentation of OpenMP constructs. See the OPARI documentation on how to instrument OpenMP source code. In addition, the application must be linked with the EPIK OpenMP (or hybrid) measurement library.

The Scalasca instrumenter command `scalasca -instrument` automatically takes care of correct compilation and linking to produce an instrumented executable, and should be prefixed to compile and link commands. Often this only requires prefixing definitions for `$(CC)` or `$(MPICC)` (and equivalents) in Makefiles. It is not necessary to prefix commands using the compiler for preprocessing, as no instrumentation is done in that case.

When compiling without the Scalasca instrumenter, the `kconfig` command can be used to simplify determining the appropriate linker flags and libraries:

```
kconfig [--mpi|--omp|--hybrid] [--for] [--user] [--32|--64] --libs
```

The `--mpi`, `--omp`, or `--hybrid` switch selects whether MPI, OpenMP or hybrid MPI/OpenMP measurement support is desired. By default, `kconfig` assumes a C or C++ program is being linked, Fortran applications have to be explicitly flagged with the

`--for` switch. With `--user`, the EPIK manual user instrumentation API can be enabled. The `--32` or `--64` switch selects the 32-bit or 64-bit version of the measurement libraries, if necessary.

Note:

A particular installation of Scalasca may not offer all measurement configurations!

The `kconfig` command can also be used to determine the right compiler flags for specifying the include directory of the `epik_user.h` or `epik_user.inc` header files when compiling without using the Scalasca instrumenter:

```
kconfig [--for] --cflags
```

or, when the user instrumentation macros should be enabled:

```
kconfig [--for] --user --cflags
```

Scalasca supports a variety of instrumentation types for user-level source routines and arbitrary regions, in addition to fully-automatic MPI and OpenMP instrumentation, as summarized in the following table:

Type	Switch	Default	Standard instrum'd routines	Other instrum'd regions	Runtime meas'ment control
MPI	—	(auto)	configured by install	—	(Sec. 4.2.2)
OpenMP	—	(auto)	(Sec. 3.5)	all parallel constructs	—
Compiler (Sec. 3.1)	`-comp`	`all`	`all` or `none`	not supported	(Sec. 4.2.1)
PDToolkit (Sec. 3.4)	`-pdt`	—	all, or selective	not supported	—
POMP (Sec. 3.3)	`-pomp`	—	manually annotated	manually annotated	—
EPIK API (Sec. 3.2)	`-user`	—	manually annotated	manually annotated	—

Table 3.1: Scalasca instrumenter option overview

When the instrumenter determines that MPI or OpenMP are being used, it automatically enables MPI library instrumentation and OPARI-based OpenMP instrumentation, respectively. The default set of instrumented MPI library functions is specified when Scalasca is installed. All OpenMP parallel constructs and API calls are instrumented by default, but instrumentation of classes of OpenMP synchronization calls can be selectively disabled as described in 3.5.

By default, automatic instrumentation of user-level source routines by the compiler is enabled (equivalent to specifying `-comp=all`). This can be disabled with `-comp=none` when desired, such as when using PDToolkit, or POMP or EPIK user API manual source annotations, enabled with `-pdt`, `-pomp` and `-user`, respectively. Compiler, PDToolkit, POMP and EPIK user API instrumentation can all be used simultaneously, or in arbitrary combinations, however, it is generally desirable to avoid instrumentation duplication (which would result if all are used to instrument the same routines).

3.1 Automatic compiler instrumentation

Most current compilers support automatic insertion of instrumentation calls at routine entry and exit(s), and Scalasca can use this capability to determine which routines are included in an instrumented measurement.

Compiler instrumentation of all routines in the specified source file(s) is enabled by default by Scalasca, or can be explicitly requested with `-comp=all`. Compiler instrumentation is disabled with `-comp=none`.

Note:

> Depending on the compiler, and how it performs instrumentation, insertion of instrumentation may disable inlining and other significant optimizations, or inlined routines may not be instrumented at all (and therefore "invisible").

Automatic compiler-based instrumentation has been tested with a number of different compilers:

- GCC (UNIX-like operating systems, not tested with Windows)
- IBM xlc, xlC (version 7 or later, IBM Blue Gene and AIX)
- IBM xlf (version 9.1 or later, IBM Blue Gene and AIX)
- PGI (Cray XT and Linux)
- Intel compilers (version 10 or later, Cray XT and Linux, not tested with Windows)
- SUN Studio compilers (Linux and Solaris, Fortran only)
- PathScale compilers (Cray XT and SiCortex)
- CCE/Cray compiler (Cray XT)
- NEC compiler (NEC SX)

In all cases, Scalasca supports automatic instrumentation of C, C++ and Fortran codes, except for the SUN Studio compilers which only provide appropriate support in their Fortran compiler.

Note:

The automatic compiler instrumentation might create a significant relative measurement overhead on short function calls. This can impact the overall application performance during measurement. C++ applications are especially prone to suffer from this, depending on application design and whether C++ STL functions are also instrumented by the compiler. Currently, it is not possible to prevent the instrumentation of specific functions on all platforms when using automatic compiler instrumentation. See Section 3.5 on how to manually instrument applications if you encounter significant overheads.

Names provided for instrumented routines depend on the compiler, which may add underscores and other decorations to Fortran and C++ routine names, and whether name "demangling" has been enabled when Scalasca was installed and could be applied successfully.

3.2 Manual region instrumentation

If the automatic compiler-based instrumentation (see Section 2.1) or semi-automatic instrumentation (see Section 3.3) procedure fails, instrumentation can be done manually. Manual instrumentation can also be used to augment automatic instrumentation with region or phase annotations, which can improve the structure of analysis reports. Generally, the main program routine should be instrumented, so that the entire execution is measured and included in the analyses.

Instrumentation can be performed in the following ways, depending on the programming language used.

Fortran:

```
#include "epik_user.inc"

subroutine foo(...)
    !declarations
    EPIK_FUNC_REG("foo")
    EPIK_USER_REG(r_name,"iteration loop")
    EPIK_FUNC_START()
    ...
    EPIK_USER_START(r_name)
    do i= 1, 100
        ...
```

```
        end do
        EPIK_USER_END(r_name)
        ...
        EPIK_FUNC_END()
    end subroutine foo
```

C/C++:

```
    #include "epik_user.h"

    void foo(...)
    {
        /* declarations */
        EPIK_USER_REG(r_name,"iteration loop");
        EPIK_FUNC_START();
        ...
        EPIK_USER_START(r_name);
        for (i = 0; i < 100; ++i)
        {
           ...
        }
        EPIK_USER_END(r_name);
        ...
        EPIK_FUNC_END();
    }
```

C++ only:

```
    #include "epik_user.h"

    void foo(...)
    {
        EPIK_TRACER("foo");
        ...
    }
```

Region identifiers (`r_name`) should be registered in each annotated function/subroutine prologue before use within the associated body, and should not already be declared in the same program scope. For C and C++, function names are automatically provided by the `EPIK_FUNC_START` and `EPIK_FUNC_END` macros (so don't need registering), whereas annotated Fortran functions and subroutines should call `EPIK_FUNC_REG` with an appropriate name.

The source files instrumented in this way have to be compiled with `-DEPIK` otherwise `EPIK_*` calls expand to nothing and are ignored. If the Scalasca instrumenter `-user` flag is used, the `EPIK` symbol will be defined automatically. Also note, that Fortran source files instrumented this way have to be preprocessed with the C preprocessor (CPP).

Manual function instrumentation in combination with automatic source-code instrumentation by the compiler leads to double instrumentation of user functions, i.e., usually only user region instrumentation is desired in this case.

For examples of how to use the EPIK user API, see the `*-epik.*` files in the example directory of the Scalasca installation.

3.3 Semi-automatic instrumentation

If you manually instrument the desired user functions and regions of your application source files using the POMP `INST` directives described below, the Scalasca instrumenter `-pomp` flag will generate instrumentation for them. POMP instrumentation directives are supported for Fortran and C/C++. The main advantages are that

- being directives, the instrumentation is ignored during "normal" compilation and
- this semi-automatic instrumentation procedure can be used when fully automatic compiler instrumentation is not supported.

The `INST BEGIN/END` directives can be used to mark any user-defined sequence of statements. If this block has several exit points (as is often the case for functions), all but the last have to be instrumented by `INST ALTEND`.

Fortran:

```
subroutine foo(...)
    !declarations
    !POMP$ INST BEGIN(foo)
    ...
    if (<condition>) then
        !POMP$ INST ALTEND(foo)
        return
    end if
    ...
    !POMP$ INST END(foo)
end subroutine foo
```

C/C++:

```
void foo(...)
```

```
{
    /* declarations */
    #pragma pomp inst begin(foo)
    ...
    if (<condition>)
    {
        #pragma pomp inst altend(foo)
        return;
    }
    ...
    #pragma pomp inst end(foo)
}
```

At least the main program function has to be instrumented in this way, and additionally, one of the following should be inserted as the first executable statement of the main program:

Fortran:

```
program main
    ! declarations
    !POMP$ INST INIT
    ...
end program main
```

C/C++:

```
int main(int argc, char** argv)
{
    /* declarations */
    #pragma pomp inst init
    ...
}
```

For an example of how to use the POMP directives, see the `*-pomp.*` files in the example directory of the Scalasca installation.

3.4 Automatic source-code instrumentation using PDT

If Scalasca has been configured with PDToolkit support, automatic source-code instrumentation can be used as an alternative instrumentation method. In this case, the source code of the target application is pre-processed before compilation, and appropriate EPIK

user API calls will be inserted automatically. However, please note that this feature is still somewhat experimental and has a number of limitations (see 3.4.1).

To enable PDT-based source-code instrumentation, call `scalasca -instrument` with the `-pdt` option, e.g.,

```
scalasca -instrument -pdt  mpicc -c foo.c
```

This will by default instrument all routines found in `foo.c`. (To avoid double instrumentation, automatic compiler instrumentation can be disabled with `-comp=none`.)

The Source-code instrumentor can also be configured to selectively instrument files and routines. For this, you need to supply the additional option

```
-optTauSelectFile=<filename>
```

after the `-pdt` option. The provided selective instrumentation file needs to be a plain text file of the following syntax:

- Empty lines are ignored; comments are introduced using a hash (#) character and reach until the end of the line.
- Files to be excluded from instrumentation can be listed in a file exclusion section. You can either list individual filenames or use the star (∗) and question mark (?) characters as wildcards. Example:

```
BEGIN_FILE_EXCLUDE_LIST
  bar.c    # Excludes file bar.c
  foo*.c   # Excludes all C files with prefix 'foo'
END_FILE_EXCLUDE_LIST
```

- To exclude certain routines from instrumentation, their names can be listed in a routine exclusion section. You can either list individual names or use the hash (#) character as a wildcard. Note that for Fortran, subroutine names must be given in all uppercase letters; for C/C++, the full function prototype including return and parameter types must be given. C functions also need to be marked with an extra capital C at the end (e.g., `"int main(int, char**) C"`). Example:

```
BEGIN_EXCLUDE_LIST
  # Exclude C function matmult
  void matmult(Matrix*, Matrix*, Matrix*) C

  # Exclude C++ functions with prefix 'sort_' and a
  # single int pointer argument
  void sort_#(int *)

  # Exclude all void functions in namespace 'foo'
  void foo::#
END_EXCLUDE_LIST
```

Unfortunately, the hash (#) character is also used for comments, so to specify a leading wildcard, place the entry in double quotes (").

For more information on how to selectively instrument code using the PDToolkit source-code instrumentor, please refer to the TAU documentation [10, 11].

3.4.1 Limitations

Since support for the PDT-based source-code instrumenter is a newly added feature, and some parts are still work in progress, a number of limitations currently exist:

- When instrumenting Fortran 77 applications, the inserted instrumentation code snippets do not yet adhere to the Fortran 77 line length limit. Typically, it is possible to work around this issue by supplying extra command line flags (e.g., `-ffixed-line-length-132` or `-qfixed=132`) to the compiler.
- If a Fortran subroutine that should be instrumented uses `len` as the name of an argument, the compilation of the instrumented code will fail (The instrumentation code uses the intrinsic function `len` which will be shadowed by the argument definition). This issue can only be resolved by renaming the argument.
- Code in C/C++ header files as well as included code in Fortran (either using the C preprocessor or the `include` keyword) will currently not be instrumented.
- Support for C++ templates and classes is currently only partially implemented.
- Advanced TAU instrumentation features such as static/dynamic timers, loop, I/O and memory instrumentation are not yet supported. Respective entries in the selective instrumentation file will be ignored.

3.5 Selective instrumentation

Scalasca experiments contain by default only summarized metrics for each callpath and process/thread. More detailed analyses, providing additional metrics regarding wait states and other inter-process inefficiencies, require that event traces are collected in buffers on each process that must be adequately sized to store events from the entire execution.

Instrumented routines which are executed frequently, while only performing a small amount of work each time they are called, have an undesirable impact on measurement. The measurement overhead for such routines is large in comparison to the execution time of the uninstrumented routine, resulting in measurement dilation. Recording these events requires significant space and analysis takes longer with relatively little improvement in quality. Filtering can be employed during measurement (described in 4.2.1) to ignore events from compiler-instrumented routines.

Ideally, such routines (or regions) should not be instrumented, to entirely remove their impact on measurement. Uninstrumented routines are still executed, but become "invisible" in measurement and subsequent analyses (as if inlined). Excess manual annotations (see Section 3.2) or POMP directives (see Section 3.3) should be removed or disabled when instrumenting.

Automatic routine instrumentation, working at the level of source modules, can be bypassed by selectively compiling such sources normally, i.e., without preprocessing with the Scalasca instrumenter.

Note:

The instrumenter is, however, still required when linking.

If only some routines within a source module should be instrumented and others left uninstrumented, the module can be split into separate files, or compiled twice with conditional preprocessor directives selecting the separate parts and producing separate object files.

Alternatively, when Scalasca has been configured with the PDToolkit, a selective instrumentation specification file can be used, as described in Section 3.4.

For OpenMP (or hybrid) applications, where there are very large numbers of synchronization operations, their instrumentation may also result in excessive measurement overhead. The OPARI tool can be instructed not to instrument any of the OpenMP synchronization constructs using `-disable sync` or a comma-separated list of specific constructs from `atomic`, `critical`, `flush`, `locks`, `master`, and `single`, e.g.,

```
scalasca -instrument  -disable atomic,locks --  gcc -fopenmp ...
```

Of course, when these constructs are not instrumented, and subsequently don't show up in measurements and analysis, the application might well still have performance problems due to too many OpenMP synchronization calls!

4 Measurement collection & analysis

The Scalasca measurement collection and analysis nexus manages the configuration and processing of performance experiments with an instrumented executable. Many different experiments can typically be performed with a single instrumented executable without needing to re-instrument, by using different measurement and analysis configurations. The default runtime summarization mode directly produces an analysis report for examination, whereas event trace collection and analysis are automatically done in two steps.

The distinctive feature of Scalasca is the automatic analysis of event traces in order to find performance bottlenecks. Internally, performance problems are specified in terms of execution patterns that represent standard situations of inefficient behavior. These patterns are used during the analysis process to recognize and quantify the inefficient behavior in the application.

The analysis of traces from OpenMP, MPI or hybrid MPI/OpenMP programs can be performed in parallel (with as many processes and threads as the original application execution), see Section 4.4. In addition, sequential analysis of traces using the KOJAK trace analyzer is still possible (see Section 4.5), although only recommended under rare circumstances.

Scalasca not only supports the analysis of function calls and user-defined source-code regions (cf. chapter 3), but also the analysis of hardware performance counter metrics, see section 4.3.

4.1 Nexus configuration

The Scalasca measurement collection and analysis nexus (SCAN) is a command prefixed to the command used to launch and run the application executable. Arguments can be given to specify whether tracing should be enabled (`-t`), a filter that should be applied (`-f filter_file`), and hardware counters that should be included in the measurement (`-m metric_list`).

The target executable is examined to determine whether MPI and/or OpenMP instrumentation is present, and the number of MPI processes and OpenMP threads are determined from the launch environment and command-line specification. These are used to generate a default name for the experiment, unless a title has been explicitly specified with `-e expt_title` (or `EPK_TITLE`). (Where the number of processes and/or threads are omitted, or were otherwise not determined, the number is replaced with the letter '`0`' is used

to indicate this.)

Note:

Configuration specified on the nexus command-line takes precedence over that specified as environment variables or in a configuration file.

Environment variables with the `SCAN_` prefix may be used to configure the nexus itself (which is a serial workflow manager process), as distinct from the instrumented application process or processes which will be measured, which are also configured via environment variables discussed in the following Section 4.2.

Serial and OpenMP programs are typically executed directly, whereas MPI (and hybrid OpenMP/MPI) programs usually require a special launcher (such as `mpiexec`) which might also specify the number of processes to be created. Many MPI launchers are automatically recognized, but if not, the MPI launcher name can be specified with the environment variable `SCAN_MPI_LAUNCHER`. When the MPI launch command is being parsed, unrecognized flags might be reported as ignored, and unrecognized options with required arguments might need to be quoted.

Note:

Launcher-specific configuration files which augment the launch command are currently not handled by Scalasca.

If the target executable isn't specified as one of the launcher arguments, it is expected to be the immediately following part of the command line. It may be necessary to use a double-dash specification (`--`) to explicitly separate the target from the preceding launcher specification.

If there is an imposter executable or script, e.g., used to specify placement, that precedes the instrumented target, it may be necessary to explcitly identify the target with the environment variable `SCAN_TARGET`.

If environment variables aren't automatically forwarded to MPI processes by the launcher, it may be necessary to specify the syntax that the launcher requires for this as `SCAN_SETENV`. For example, if an environment variable VAR and value VAL must be exported with "`--`export VAR VAL" use `SCAN_SETENV=--export`, or use `SCAN_SETENV=-setenv=` for "-setenv VAR=VAL" syntax.

Trace analysis is done with different trace analyzers according to availability and the type of experiment. An alternate trace analyzer can be specified with `SCAN_TRACE_ANALYZER`. Options to be given to the trace analyzer can be specified with `SCAN_ANALYZE_OPTS`. Trace data can be automatically removed after successful trace analysis by setting `SCAN_CLEAN`.

4.2 Measurement configuration

A number of configuration variables can be used to control the EPIK measurement runtime configuration: for an annotated list of configuration variables, and their current settings, run the `epik_conf` command. Configuration variables can be specified via environment variables or in a configuration file called `EPIK.CONF`: by default the current directory is searched for this file, or an alternative location can be specified with the `EPK_CONF` environment variable.

The values for configuration variables can contain (sub)strings of the form `$XYZ` or `${XYZ}` where `XYZ` is the name of another configuration variable. Evaluation of the configuration variable is done at runtime when measurement is initiated.

When tracing (large-scale) MPI applications it is recommended to set the `EPK_LDIR` and `EPK_GDIR` variables to the same location, as in such cases intermediate file writing is avoided and can greatly improve performance. Therefore, this is the default setting.

4.2.1 Compiler-instrumented routine filtering

When automatic compiler instrumentation has been used to instrument user-level source-program routines (classified as USR routines), there are cases where measurement and associated analysis are degraded, e.g., by small, frequently-executed and/or generally uninteresting functions, methods and subroutines.

A measurement filtering capability is therefore supported for most (but not all) compilers. A file containing the names of functions (one per line) to be excluded from measurement can be specified using the EPIK configuration variable `EPK_FILTER` or alternatively via the `-f <filter_file>` option of the `scalasca -analyze` command (and will be archived in `epik_<title>/epik.filt` as part of the experiment). Filter function names can include wildcards ("`*`") and, if name demangling is not supported, then linker names must be used. On the other hand, if C++ name demangling is supported, "`*`" characters indicating pointer variables have to be escaped using a backslash.

Whenever a function marked for filtering is executed, the measurement library skips making a measurement event, thereby substantially reducing the overhead and impact of such functions. In some cases, even this minimal instrumentation processing may be undesirable, and the function should be excluded from instrumentation as described in Section 3.5.

4.2.2 Selective MPI event generation

The Message Passing Intercace (MPI) adapter of EPIK supports the tracing of most of MPI's 300+ function calls. MPI defines a so-called 'profiling interface' that supports the provision of wrapper libraries that can easily interposed between the user application and

Token	Module
ALL	Activate all available modules
DEFAULT	Activate the configured default modules of CG:COLL:ENV:IO:P2P:RMA:TOPO. This can be used to easily activate additional modules.
CG	Communicators and groups
COLL	Collective communication
ENV	Environmental management
ERR	Error handlers
EXT	External interfaces
IO	I/O
MISC	Miscellaneous
P2P	Point-to-point communication
RMA	One-sided communication
SPAWN	Process management interface (aka Spawn)
TOPO	Topology communicators
TYPE	MPI Datatypes

the MPI library calls.

EPIK supports selective event generation. Currently, this means that at start time of the application, the user can decide whether event generation is turned on or off for a group of functions. These groups are the listed sub modules of this adapter. Each module has a short string token that identifies this group. To activate event generation for a specific group, the user can specify a colon-seperated list of tokens in the configuration variable `EPK_MPI_ENABLED`. Additionally, special tokens exist to ease the handling by the user. A complete list of available tokens that can be specified in the runtime configuration is listed below.

Note:

> Event generation in this context only relates to flow and and transfer events. Tracking of communicators, groups, and other internal data is unaffected and always turned on.

Example:

```
EPK_MPI_ENABLED=ENV:P2P
```

This will enable event generation for environmental managment, including `MPI_Init` and `MPI_Finalize`, as well as point-to-point communication, but will disable it for all other functions groups.

A shorthand to get event generation for all supported function calls is

```
EPK_MPI_ENABLED=ALL
```

A shorthand to add a single group, e.g. `TYPE`, to the configured default is

```
EPK_MPI_ENABLED=DEFAULT:TYPE
```

A detailed overview of the MPI functions associated with each group can be found in Appendix A.

4.3 Measurement and analysis of hardware counter metrics

If the Scalasca measurement library EPIK has been built with hardware counter support enabled (see `INSTALL` file), it is capable of processing hardware counter information as part of event handling. (This can be checked by running `epik_conf` and seeing whether `EPK_METRICS_SPEC` is set.)

Counters are processed into counter metrics during runtime summarization, and recorded as part of event records in collected traces. Note that the number of counters recorded determines measurement and analysis overheads, as well as the sizes of measurement storage datastructures, event traces and analysis reports. Counter metrics recorded in event traces are currently ignored by the Scalasca parallel trace analyzer, and it is generally recommended that they should only be specified for summarization measurements.

To request the measurement of certain counters, set the variable `EPK_METRICS` to a colon-separated list of counter names, or a predefined platform-specific group. Alternatively specify the desired metrics with `-m <metriclist>` argument to the Scalasca measurement collection and analysis system (`scalasca -analyze`). Hardware counter measurement is disabled by default.

Metric names can be chosen from the list contained in file `doc/METRICS.SPEC` or may be PAPI preset names or platform-specific "native" counter names. `METRICS.SPEC` also contains specifications of groups of (related) counters which may conveniently be measured simultaneously on various platforms. The installed `doc/METRICS.SPEC` specification is overridden by a file named `METRICS.SPEC` in the current working directory, or specified by the EPIK configuration variable `EPK_METRICS_SPEC`.

If any of the requested counters are not recognized or the full list of counters cannot be recorded due to hardware-resource limits, measurement of the program execution will be aborted with an error message.

Counter metrics appear in the Performance Metrics pane of the CUBE3 browser. Relationships between counter metrics which define hierarchies are also specified in the file `METRICS.SPEC` — those without specified relationships are listed separately.

Experiments with subsets of the counter metrics required for a full hierarchy could previously be combined into composite experiments using the `cube_merge` utility. Note that a replacement for this utility is still under development and not yet available. Generally several measurement experiments are required, and the groupings of counters provided in METRICS.SPEC can act as a guide for these.

The default `doc/METRICS.SPEC` provides generic metric specifications which can be used for analysis on any platform. Additional platform-specific example metric specifications are provided in the examples directory. If desired, an example `METRICS.SPEC` appropriate for the platform where the measurements will be (or have been) recorded can be used instead of the default `doc/METRICS.SPEC` via setting the `EPK_METRICS_SPEC` configuration variable or replacing the installed file.

EXPERT analysis (see Section 4.5) can further be customized using additional environment variables: `EPT_INCOMPLETE_COMPUTATION` can be set to accept metric computations which are missing one or more component measurement (while not generally useful on its own, it can allow more detailed metric hierarchies to be created when experiments are combined); `EPT_MEASURED_METRICS` modifies the handling of unparented measured metrics, such that they can be ignored (value 0), listed separately (value 1, the default) or listed together with parented metrics (value 2).

4.4 Automatic parallel event trace analysis

SCOUT is Scalasca's automatic analyzer for EPIK event traces. It is used internally by the Scalasca measurement collection and analysis nexus (`scalasca -analyze`) when event tracing is configured, or can be explicitly executed on event traces in EPIK measurement archives. Depending on the build configuration and the capabilities of the target platform, SCOUT may be available in four forms:

scout.ser is always built. It is used to analyze event traces generated by serial applications. It can also be used to analyze event traces from pure OpenMP applications, however, it will then only provide information about the master thread.

scout.omp is built whenever Scalasca is configured with OpenMP support. It is used to analyze event traces generated by pure OpenMP applications. It can also be used to analyze event traces from serial applications.

scout.mpi is built whenever Scalasca is configured with MPI support. It is used to analyze event traces generated by pure MPI applications. It can also be used on traces from hybrid MPI/OpenMP applications, however, it will then only provide information about the master thread of each process and its MPI activities.

scout.hyb is built if Scalasca is configured with hybrid MPI/OpenMP support. It is used to analyze event traces generated by hybrid MPI/OpenMP applications, providing information about all OpenMP threads of each MPI process.

The appropriate SCOUT variant can be explicitly executed on event traces in EPIK measurement archives using

```
$MPIEXEC $MPIEXEC_FLAGS <scout.type> [-s] epik_<title>
```

which produces an (intermediate) analysis report `epik_<title>/scout.cube`.

Event traces collected on clusters without a synchronized clock may contain logical clock condition violations [2] (such as a receive completing before the corresponding send is initiated). When SCOUT detects this, it reports a warning that the analysis may be inconsistent and recommends (re-)running trace analysis with its integrated timestamp synchronization algorithm (based on the controlled logical clock [1]) activated: this auxilliary trace processing is specified with the optional `-s` flag to SCOUT.

Alternatively, event trace analysis can be (re-)initiated using the `scalasca -analyze` command, e.g.,

```
scalasca -analyze -a -e epik_<title> $MPIEXEC $MPIEXEC_FLAGS
```

where `MPIEXEC` is the command used to configure and launch MPI applications, and is typically identical to that used to launch the user MPI application. In the second case, the `scalasca -analyze` command will automatically figure out which SCOUT variant should be used and/or is available. To activate the integrated timestamp synchronization algorithm when using the `scalasca -analyze` command, the environment variable `SCAN_ANALYZE_OPTS` needs to be set to `-s`.

Note:

The number of MPI processes for SCOUT must be identical to the number of MPI processes for the original application! Furthermore, if SCOUT is executed on OpenMP or hybrid MPI/OpenMP traces, it is recommended to set the environment variable `OMP_NUM_THREADS` to the value used for the original application, although SCOUT will automatically try to create the appropriate number of OpenMP threads.

Warning:

The `scout.omp` and `scout.hyb` analyzer require pure OpenMP and hybrid OpenMP/MPI applications to use the same number of threads during all parallel regions. A dynamically changing number of threads is not supported, and typically will result in deadlock! However, different numbers of threads on each process are supported.

When running the SCOUT analyzer on (back-end) compute nodes with a different architecture to their system front-end, remember to specify the path to the appropriate (back-end) version (e.g., `$SCALASCA_RTS/<scout.type>`).

If your MPI library doesn't automatically support passing command-line arguments to all MPI processes, the name of the experiment to analyze may need to be passed in a special

form (e.g., `-args "epik_<title>"`) or can be specified via the `EPK_TITLE` configuration variable (in a `EPIK.CONF` file or set in the environment for each MPI process, e.g., `-env "EPK_TITLE=<title>"`).

Note:

SCOUT processes may require two times the memory of the largest MPI-rank trace (as reported as `max_tbc` by `scalasca -examine -s` or `cube3_score`) to complete analysis without paging to disk. Hardware counters recorded in event traces are currently ignored by SCOUT, however, hardware counter metrics can be found in the runtime summarization analysis report (summary.cube) which is also produced by default when tracing is enabled.

4.5 Automatic sequential event trace analysis

EXPERT is an automatic serial analyzer for *merged* EPILOG event traces. It can be manually applied to OpenMP, MPI and hybrid MPI/OpenMP traces in EPIK experiment archives after they have been merged via

```
elg_merge epik_<title>
```

to produce `epik_<title>/epik.elg`.

Note:

It may take quite a long time to merge large event traces, and the resulting `epik.elg` will typically be more than three times as large as the unmerged process traces!

Explicit execution of EXPERT on a merged EPILOG event trace in an EPIK experiment archive via

```
expert epik_<title>
```

produces an analysis report `epik_<title>/expert.cube`.

Note:

Bear in mind, that both merging of MPI rank traces and EXPERT analysis are sequential operations that might take a long time for large experiments!

Warning:

The EXPERT analyzer requires the event trace to represent a call tree with a single root. Therefore you should instrument the entry and exit of the application's "main" function if necessary. Also note that EXPERT requires OpenMP applications to use the same number of threads during all parallel regions. A dynamically changing number of threads is not supported!

Integrated merged trace analysis and results presentation is provided by the command:

```
kanal epik_<title>
```

or

```
kanal <file>[.elg|.cube]
```

The command takes as argument either an EPIK experiment archive (containing a merged trace), a merged trace <file>.elg or a generated analysis report <file>.cube. If <file>.cube already exists (and is newer than <file>.elg), CUBE3 is used to present it and browse the analysis. If the trace <file>.elg is newer (or no analysis file exists), then EXPERT is run to generate <file>.cube before it is presented with CUBE3. Where generation of a new <file>.cube would overwrite an existing (older) file with the same name, a prompt will confirm whether to continue.

The EXPERT event trace analysis and CUBE analysis visualization can also be executed separately, which is particularly appropriate when the CUBE viewer is installed on a separate system (e.g., desktop) from the measurement system (e.g., a remote HPC system).

EXPERT analysis performance for particular trace files can be tuned via EARL environment variables which trade efficiency and memory requirements. In order to analyze a trace file, EXPERT reads the trace file once from the beginning to the end. After accessing a particular event, EXPERT might request other events usually from the recent past of the event or ask for state information related to one of those events. Random access to events as well as the calculation of state information is done inside the EARL event accessor library, a component used by EXPERT.

During the analysis process, EARL dynamically builds up a sparse index structure on the trace file. At fixed intervals the state information is stored in so-called bookmarks to speed up random access to events. If a particular event is requested, EARL usually needs not start reading from the beginning of the trace file in order to find it. Instead, the interpreter looks for the nearest bookmark and takes the state information from there which is required to correctly interpret the subsequent events from the file. Then it starts reading the trace from there until it reaches the desired event. The distance of bookmarks can be set using the following environment variable:

```
EARL_BOOKMARK_DISTANCE (default 10000)
```

To gain further efficiency, EARL automatically caches the most recently processed events in a history buffer. The history buffer always contains a contiguous subsequence of the event trace and the state information referring to the beginning of this subsequence. So all information related to events in the history buffer can be completely generated from the buffer including state information. The size of the history buffer can be set using another environment variable:

```
EARL_HISTORY_SIZE (default 1000 * number of processes or threads)
```

Note:

Choosing the right buffer parameters is usually a trade-off decision between access efficiency and memory requirements. In particular, for very long traces with many events or very wide traces with many processes or threads, adjustment of these parameters might be recommended.

5 Additional utilities

5.1 Additional EPILOG event trace utilities

Process-local EPILOG traces in EPIK experiment archives can be *merged* by executing

```
elg_merge epik_<title>
```

in order to produce a single merged trace file `epik_<title>/epik.elg`.

Note:

It may take quite a long time to merge large event traces, and the resulting `epik.elg` will typically be more than three times as large as the unmerged process traces!

Two utility programs are provided to check the *correctness* and to *summarize* the contents of EPILOG trace files:

```
elg_print <file>.elg
```

Prints the contents of the EPILOG trace file `<file>.elg` to the standard output stream. `elg_print` creates a readable representation of the EPILOG low-level record format. This is mainly provided for debugging purposes to check the correct structure and content of the EPILOG trace records.

```
elg_stat <file>.elg
```

By default, `elg_stat` calculates and reports some very simple event statistics to standard output. In addition, options `-d` (definition records) and `-e` (event records) enable the printing of a human-readable representation of the trace contents on the event level.

5.2 Trace converters

The following utility programs can be used to *convert* merged EPILOG trace file into other formats.

If support for the trace formats OTF and/or VTF3 were included during configuration and installation, merged EPILOG event traces can be converted for visual analysis with the VAMPIR trace visualizer from TU Dresden ZIH [6].

To convert a merged trace to the VampirTrace Open Trace Format (OTF) use

```
elg2otf epik_<title>
```

and to convert to the older VAMPIR version 3 format (VTF3) use

```
elg2vtf3 epik_<title>/<file>.elg
```

which stores the resulting trace in `<file>.vpt`.

Experimental support is provided to convert merged EPILOG traces to the format used by the PARAVER trace visualizer from the Barcelona Supercomputing Center [3] via

```
elg2prv epik_<title>
```

and to the Slog2 format used by MPE from Argonne National Laboratory [4] via

```
elgTOslog2 epik_<title>/epik.elg
```

To visualize the resulting Slog2 file with Jumpshot use

```
jumpshot epik_<title>/epik.elg.slog2
```

5.3 Recording user-specified virtual topologies

A virtual topology defines the mapping of processes and threads onto the application domain, such as a weather model simulation area. In general, a virtual topology is specified as a graph (e.g., a ring) or a Cartesian topology such as two- or higher-dimensional grids. Virtual topologies can include processes, threads or a combination of both, depending on the programming model.

Virtual topologies can be useful to identify performance problems. Mapping performance data onto the topology can help uncover inefficient interactions between neighbors and suggest algorithmic improvements. EPIK supports the recording of 3D Cartesian grids as the most common case. To do this, the user has two options:

1. using MPI Cartesian-topology functions
2. manual recording using the EPILOG topology API

If an application uses MPI topology functions to set up a Cartesian grid, EPIK automatically includes this information in the measurement experiment.

In addition, EPIK provides users who do not use MPI topologies with an API to define a one-, two- or three- dimensional Cartesian topology. These functions are available in C and Fortran. To use the C API, include `elg_topol.h`. Whereas in C all functions start with the prefix `elg_`, they start with `elgf_` in Fortran. Here are the signatures of these two functions:

1. `elg(f)_cart_create(size_0, size_1, size_2, period_0, period_1, period_2);`

   defines a Cartesian grid topology of up to 3 dimensions where

   - `size_X` is an integer describing the number of locations in dimension X and
   - `period_X` is an integer describing the periodicity in dimension X. It should be
     - *zero* if dimension is not periodic, and
     - *non-zero* if dimension is periodic.

   To specify a grid with less than three dimensions, set `size_X` to zero for the dimensions not being used.

2. `elg(f)_cart_coords(coord_0, coord_1, coord_2);`

   defines the coordinates of the calling process or thread (i.e., location) in the previously defined grid where

   - `coord_X` is an integer describing the coordinate of a location in dimension `X` with $X \in [0, \text{size_X} - 1]$

   This function must be called exactly once by every process/thread that is part of the Cartesian topology.

Note:

There are currectly a few restrictions that need to be obeyed when using the EPILOG topology API:

- Manual topologies need to be defined via the API *before* any MPI Cartesian-topology function is called. Manual topologies defined afterwards will be ignored.
- For each manually defined topology, every MPI process has to call the creation function `elg(f)_cart_create()` *exactly once*. However, it doesn't matter which thread executes the call.
- When defining multiple topologies using the EPILOG API, all coordinates of a topology need to defined using `elg(f)_cart_coords()` *before* defining the next topology using `elg(f)_cart_create()`.

A MPI wrapper affiliation

Some wrapper functions are affiliated with a function group that has not been described for direct user access in section 4.2.2. These groups are subgroups that contain function calls that are only enabled when both main groups are enabled. The reason for this is to control the amount of events generated during measurement, a user might want to turn off the measurement of non-critical function calls before the measurement of the complete main group is turned off. Subgroups can either be related to `MISC` (miscellaneous functions, e.g. handle conversion), `EXT` (external interfaces, e.g. handle attributes), or `ERR` (error handlers).

For example, the functions in group `CG_MISC` will only generate events if both groups `CG` and `MISC` are enabled at runtime.

A.1 Function to group

Function	Group
MPI_Abort	EXT
MPI_Accumulate	RMA
MPI_Add_error_class	ERR
MPI_Add_error_code	ERR
MPI_Add_error_string	ERR
MPI_Address	MISC
MPI_Allgather	COLL
MPI_Allgatherv	COLL
MPI_Alloc_mem	MISC
MPI_Allreduce	COLL
MPI_Alltoall	COLL
MPI_Alltoallv	COLL
MPI_Alltoallw	COLL
MPI_Attr_delete	CG_EXT
MPI_Attr_get	CG_EXT
MPI_Attr_put	CG_EXT
MPI_Barrier	COLL

MPI_Bcast	COLL
MPI_Bsend	P2P
MPI_Bsend_init	P2P
MPI_Buffer_attach	P2P
MPI_Buffer_detach	P2P
MPI_Cancel	P2P
MPI_Cart_coords	TOPO
MPI_Cart_create	TOPO
MPI_Cart_get	TOPO
MPI_Cart_map	TOPO
MPI_Cart_rank	TOPO
MPI_Cart_shift	TOPO
MPI_Cart_sub	TOPO
MPI_Cartdim_get	TOPO
MPI_Close_port	SPAWN
MPI_Comm_accept	SPAWN
MPI_Comm_c2f	CG_MISC
MPI_Comm_call_errhandler	CG_ERR
MPI_Comm_compare	CG
MPI_Comm_connect	SPAWN
MPI_Comm_create	CG
MPI_Comm_create_errhandler	CG_ERR
MPI_Comm_create_keyval	CG_EXT
MPI_Comm_delete_attr	CG_EXT
MPI_Comm_disconnect	SPAWN
MPI_Comm_dup	CG
MPI_Comm_f2c	CG_MISC
MPI_Comm_free	CG
MPI_Comm_free_keyval	CG_EXT
MPI_Comm_get_attr	CG_EXT
MPI_Comm_get_errhandler	CG_ERR
MPI_Comm_get_name	CG_EXT
MPI_Comm_get_parent	SPAWN
MPI_Comm_group	CG
MPI_Comm_join	SPAWN
MPI_Comm_rank	CG
MPI_Comm_remote_group	CG
MPI_Comm_remote_size	CG
MPI_Comm_set_attr	CG_EXT

MPI_Comm_set_errhandler	CG_ERR
MPI_Comm_set_name	CG_EXT
MPI_Comm_size	CG
MPI_Comm_spawn	SPAWN
MPI_Comm_spawn_multiple	SPAWN
MPI_Comm_split	CG
MPI_Comm_test_inter	CG
MPI_Dims_create	TOPO
MPI_Dist_graph_create	TOPO
MPI_Dist_graph_create_adjacent	TOPO
MPI_Dist_graph_neighbors	TOPO
MPI_Dist_graph_neighbors_count	TOPO
MPI_Errhandler_create	ERR
MPI_Errhandler_free	ERR
MPI_Errhandler_get	ERR
MPI_Errhandler_set	ERR
MPI_Error_class	ERR
MPI_Error_string	ERR
MPI_Exscan	COLL
MPI_File_c2f	IO_MISC
MPI_File_call_errhandler	IO_ERR
MPI_File_close	IO
MPI_File_create_errhandler	IO_ERR
MPI_File_delete	IO
MPI_File_f2c	IO_MISC
MPI_File_get_amode	IO
MPI_File_get_atomicity	IO
MPI_File_get_byte_offset	IO
MPI_File_get_errhandler	IO_ERR
MPI_File_get_group	IO
MPI_File_get_info	IO
MPI_File_get_position	IO
MPI_File_get_position_shared	IO
MPI_File_get_size	IO
MPI_File_get_type_extent	IO
MPI_File_get_view	IO
MPI_File_iread	IO
MPI_File_iread_at	IO
MPI_File_iread_shared	IO

MPI_File_iwrite	IO
MPI_File_iwrite_at	IO
MPI_File_iwrite_shared	IO
MPI_File_open	IO
MPI_File_preallocate	IO
MPI_File_read	IO
MPI_File_read_all	IO
MPI_File_read_all_begin	IO
MPI_File_read_all_end	IO
MPI_File_read_at	IO
MPI_File_read_at_all	IO
MPI_File_read_at_all_begin	IO
MPI_File_read_at_all_end	IO
MPI_File_read_ordered	IO
MPI_File_read_ordered_begin	IO
MPI_File_read_ordered_end	IO
MPI_File_read_shared	IO
MPI_File_seek	IO
MPI_File_seek_shared	IO
MPI_File_set_atomicity	IO
MPI_File_set_errhandler	IO_ERR
MPI_File_set_info	IO
MPI_File_set_size	IO
MPI_File_set_view	IO
MPI_File_sync	IO
MPI_File_write	IO
MPI_File_write_all	IO
MPI_File_write_all_begin	IO
MPI_File_write_all_end	IO
MPI_File_write_at	IO
MPI_File_write_at_all	IO
MPI_File_write_at_all_begin	IO
MPI_File_write_at_all_end	IO
MPI_File_write_ordered	IO
MPI_File_write_ordered_begin	IO
MPI_File_write_ordered_end	IO
MPI_File_write_shared	IO
MPI_Finalize	ENV
MPI_Finalized	ENV

MPI_Free_mem	MISC
MPI_Gather	COLL
MPI_Gatherv	COLL
MPI_Get	RMA
MPI_Get_address	MISC
MPI_Get_count	EXT
MPI_Get_elements	EXT
MPI_Get_processor_name	EXT
MPI_Get_version	MISC
MPI_Graph_create	TOPO
MPI_Graph_get	TOPO
MPI_Graph_map	TOPO
MPI_Graph_neighbors	TOPO
MPI_Graph_neighbors_count	TOPO
MPI_Graphdims_get	TOPO
MPI_Grequest_complete	EXT
MPI_Grequest_start	EXT
MPI_Group_c2f	CG_MISC
MPI_Group_compare	CG
MPI_Group_difference	CG
MPI_Group_excl	CG
MPI_Group_f2c	CG_MISC
MPI_Group_free	CG
MPI_Group_incl	CG
MPI_Group_intersection	CG
MPI_Group_range_excl	CG
MPI_Group_range_incl	CG
MPI_Group_rank	CG
MPI_Group_size	CG
MPI_Group_translate_ranks	CG
MPI_Group_union	CG
MPI_Ibsend	P2P
MPI_Info_c2f	MISC
MPI_Info_create	MISC
MPI_Info_delete	MISC
MPI_Info_dup	MISC
MPI_Info_f2c	MISC
MPI_Info_free	MISC
MPI_Info_get	MISC

MPI_Info_get_nkeys	MISC
MPI_Info_get_nthkey	MISC
MPI_Info_get_valuelen	MISC
MPI_Info_set	MISC
MPI_Init	ENV
MPI_Init_thread	ENV
MPI_Initialized	ENV
MPI_Intercomm_create	CG
MPI_Intercomm_merge	CG
MPI_Iprobe	P2P
MPI_Irecv	P2P
MPI_Irsend	P2P
MPI_Is_thread_main	ENV
MPI_Isend	P2P
MPI_Issend	P2P
MPI_Keyval_create	CG_EXT
MPI_Keyval_free	CG_EXT
MPI_Lookup_name	SPAWN
MPI_Op_c2f	MISC
MPI_Op_commutative	MISC
MPI_Op_create	MISC
MPI_Op_f2c	MISC
MPI_Op_free	MISC
MPI_Open_port	SPAWN
MPI_Pack	TYPE
MPI_Pack_external	TYPE
MPI_Pack_external_size	TYPE
MPI_Pack_size	TYPE
MPI_Pcontrol	PERF
MPI_Probe	P2P
MPI_Publish_name	SPAWN
MPI_Put	RMA
MPI_Query_thread	ENV
MPI_Recv	P2P
MPI_Recv_init	P2P
MPI_Reduce	COLL
MPI_Reduce_local	COLL
MPI_Reduce_scatter	COLL
MPI_Reduce_scatter_block	COLL

MPI_Register_datarep	IO
MPI_Request_c2f	MISC
MPI_Request_f2c	MISC
MPI_Request_free	P2P
MPI_Request_get_status	MISC
MPI_Rsend	P2P
MPI_Rsend_init	P2P
MPI_Scan	COLL
MPI_Scatter	COLL
MPI_Scatterv	COLL
MPI_Send	P2P
MPI_Send_init	P2P
MPI_Sendrecv	P2P
MPI_Sendrecv_replace	P2P
MPI_Sizeof	TYPE
MPI_Ssend	P2P
MPI_Ssend_init	P2P
MPI_Start	P2P
MPI_Startall	P2P
MPI_Status_c2f	MISC
MPI_Status_f2c	MISC
MPI_Status_set_cancelled	EXT
MPI_Status_set_elements	EXT
MPI_Test	P2P
MPI_Test_cancelled	P2P
MPI_Testall	P2P
MPI_Testany	P2P
MPI_Testsome	P2P
MPI_Topo_test	TOPO
MPI_Type_c2f	TYPE_MISC
MPI_Type_commit	TYPE
MPI_Type_contiguous	TYPE
MPI_Type_create_darray	TYPE
MPI_Type_create_f90_complex	TYPE
MPI_Type_create_f90_integer	TYPE
MPI_Type_create_f90_real	TYPE
MPI_Type_create_hindexed	TYPE
MPI_Type_create_hvector	TYPE
MPI_Type_create_indexed_block	TYPE

MPI_Type_create_keyval	TYPE_EXT
MPI_Type_create_resized	TYPE
MPI_Type_create_struct	TYPE
MPI_Type_create_subarray	TYPE
MPI_Type_delete_attr	TYPE_EXT
MPI_Type_dup	TYPE
MPI_Type_extent	TYPE
MPI_Type_f2c	TYPE_MISC
MPI_Type_free	TYPE
MPI_Type_free_keyval	TYPE_EXT
MPI_Type_get_attr	TYPE_EXT
MPI_Type_get_contents	TYPE
MPI_Type_get_envelope	TYPE
MPI_Type_get_extent	TYPE
MPI_Type_get_name	TYPE_EXT
MPI_Type_get_true_extent	TYPE
MPI_Type_hindexed	TYPE
MPI_Type_hvector	TYPE
MPI_Type_indexed	TYPE
MPI_Type_lb	TYPE
MPI_Type_match_size	TYPE
MPI_Type_set_attr	TYPE_EXT
MPI_Type_set_name	TYPE_EXT
MPI_Type_size	TYPE
MPI_Type_struct	TYPE
MPI_Type_ub	TYPE
MPI_Type_vector	TYPE
MPI_Unpack	TYPE
MPI_Unpack_external	TYPE
MPI_Unpublish_name	SPAWN
MPI_Wait	P2P
MPI_Waitall	P2P
MPI_Waitany	P2P
MPI_Waitsome	P2P
MPI_Win_c2f	RMA_MISC
MPI_Win_call_errhandler	RMA_ERR
MPI_Win_complete	RMA
MPI_Win_create	RMA
MPI_Win_create_errhandler	RMA_ERR

MPI_Win_create_keyval	RMA_EXT
MPI_Win_delete_attr	RMA_EXT
MPI_Win_f2c	RMA_MISC
MPI_Win_fence	RMA
MPI_Win_free	RMA
MPI_Win_free_keyval	RMA_EXT
MPI_Win_get_attr	RMA_EXT
MPI_Win_get_errhandler	RMA_ERR
MPI_Win_get_group	RMA
MPI_Win_get_name	RMA_EXT
MPI_Win_lock	RMA
MPI_Win_post	RMA
MPI_Win_set_attr	RMA_EXT
MPI_Win_set_errhandler	RMA_ERR
MPI_Win_set_name	RMA_EXT
MPI_Win_start	RMA
MPI_Win_test	RMA
MPI_Win_unlock	RMA
MPI_Win_wait	RMA
MPI_Wtick	EXT
MPI_Wtime	EXT

A.2 Group to function

CG	Communicators and Groups
	MPI_Comm_compare, MPI_Comm_create, MPI_Comm_dup, MPI_Comm_free, MPI_Comm_group, MPI_Comm_rank, MPI_Comm_remote_group, MPI_Comm_remote_size, MPI_Comm_size, MPI_Comm_split, MPI_Comm_test_inter, MPI_Group_compare, MPI_Group_difference, MPI_Group_excl, MPI_Group_free, MPI_Group_incl, MPI_Group_intersection, MPI_Group_range_excl, MPI_Group_range_incl, MPI_Group_rank, MPI_Group_size, MPI_Group_translate_ranks, MPI_Group_union, MPI_Intercomm_create, MPI_Intercomm_merge,

CG_ERR	**Error handlers for Communicators and Groups**
	MPI_Comm_call_errhandler, MPI_Comm_create_errhandler, MPI_Comm_get_errhandler, MPI_Comm_set_errhandler,

CG_EXT	**External interfaces for Communicators and Groups**
	MPI_Attr_delete, MPI_Attr_get, MPI_Attr_put, MPI_Comm_create_keyval, MPI_Comm_delete_attr, MPI_Comm_free_keyval, MPI_Comm_get_attr, MPI_Comm_get_name, MPI_Comm_set_attr, MPI_Comm_set_name, MPI_Keyval_create, MPI_Keyval_free,

CG_MISC	**Miscellaneous functions for Communicators and Groups**
	MPI_Comm_c2f, MPI_Comm_f2c, MPI_Group_c2f, MPI_Group_f2c,

COLL	**Collective communication**
	MPI_Allgather, MPI_Allgatherv, MPI_Allreduce, MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw, MPI_Barrier, MPI_Bcast, MPI_Exscan, MPI_Gather, MPI_Gatherv, MPI_Reduce, MPI_Reduce_local, MPI_Reduce_scatter, MPI_Reduce_scatter_block, MPI_Scan, MPI_Scatter, MPI_Scatterv,

ENV	**Environmental management**
	MPI_Finalize, MPI_Finalized, MPI_Init, MPI_Init_thread, MPI_Initialized, MPI_Is_thread_main, MPI_Query_thread,

ERR	**Common error handlers**
	MPI_Add_error_class, MPI_Add_error_code, MPI_Add_error_string, MPI_Errhandler_create, MPI_Errhandler_free, MPI_Errhandler_get, MPI_Errhandler_set, MPI_Error_class, MPI_Error_string,

EXT	**Common external interfaces**
	MPI_Abort, MPI_Get_count, MPI_Get_elements, MPI_Get_processor_name, MPI_Grequest_complete, MPI_Grequest_start, MPI_Status_set_cancelled, MPI_Status_set_elements, MPI_Wtick, MPI_Wtime,

IO	**Parallel I/O**
	MPI_File_close, MPI_File_delete, MPI_File_get_amode, MPI_File_get_atomicity, MPI_File_get_byte_offset, MPI_File_get_group, MPI_File_get_info, MPI_File_get_position, MPI_File_get_position_shared, MPI_File_get_size, MPI_File_get_type_extent, MPI_File_get_view, MPI_File_iread, MPI_File_iread_at, MPI_File_iread_shared, MPI_File_iwrite, MPI_File_iwrite_at, MPI_File_iwrite_shared, MPI_File_open, MPI_File_preallocate, MPI_File_read, MPI_File_read_all, MPI_File_read_all_begin, MPI_File_read_all_end, MPI_File_read_at, MPI_File_read_at_all, MPI_File_read_at_all_begin, MPI_File_read_at_all_end, MPI_File_read_ordered, MPI_File_read_ordered_begin, MPI_File_read_ordered_end, MPI_File_read_shared, MPI_File_seek, MPI_File_seek_shared, MPI_File_set_atomicity, MPI_File_set_info, MPI_File_set_size, MPI_File_set_view, MPI_File_sync, MPI_File_write, MPI_File_write_all, MPI_File_write_all_begin, MPI_File_write_all_end, MPI_File_write_at, MPI_File_write_at_all, MPI_File_write_at_all_begin, MPI_File_write_at_all_end, MPI_File_write_ordered, MPI_File_write_ordered_begin, MPI_File_write_ordered_end, MPI_File_write_shared, MPI_Register_datarep,

IO_ERR	Error handlers for Parallel I/O
	MPI_File_call_errhandler, MPI_File_create_errhandler, MPI_File_get_errhandler, MPI_File_set_errhandler,

IO_MISC	Miscellaneous functions for Parallel I/O
	MPI_File_c2f, MPI_File_f2c,

MISC	Miscellaneous functions
	MPI_Address, MPI_Alloc_mem, MPI_Free_mem, MPI_Get_address, MPI_Get_version, MPI_Info_c2f, MPI_Info_create, MPI_Info_delete, MPI_Info_dup, MPI_Info_f2c, MPI_Info_free, MPI_Info_get, MPI_Info_get_nkeys, MPI_Info_get_nthkey, MPI_Info_get_valuelen, MPI_Info_set, MPI_Op_c2f, MPI_Op_commutative, MPI_Op_create, MPI_Op_f2c, MPI_Op_free, MPI_Request_c2f, MPI_Request_f2c, MPI_Request_get_status, MPI_Status_c2f, MPI_Status_f2c,

P2P	Point-to-point communication
	MPI_Bsend, MPI_Bsend_init, MPI_Buffer_attach, MPI_Buffer_detach, MPI_Cancel, MPI_Ibsend, MPI_Iprobe, MPI_Irecv, MPI_Irsend, MPI_Isend, MPI_Issend, MPI_Probe, MPI_Recv, MPI_Recv_init, MPI_Request_free, MPI_Rsend, MPI_Rsend_init, MPI_Send, MPI_Send_init, MPI_Sendrecv, MPI_Sendrecv_replace, MPI_Ssend, MPI_Ssend_init, MPI_Start, MPI_Startall, MPI_Test, MPI_Test_cancelled, MPI_Testall, MPI_Testany, MPI_Testsome, MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome,

PERF	**Profiling Interface**
	MPI_Pcontrol,

RMA	**One-sided communication (Remote Memory Access)**
	MPI_Accumulate, MPI_Get, MPI_Put, MPI_Win_complete, MPI_Win_create, MPI_Win_fence, MPI_Win_free, MPI_Win_get_group, MPI_Win_lock, MPI_Win_post, MPI_Win_start, MPI_Win_test, MPI_Win_unlock, MPI_Win_wait,

RMA_ERR	**Error handlers for One-sided communication (Remote Memory Access)**
	MPI_Win_call_errhandler, MPI_Win_create_errhandler, MPI_Win_get_errhandler, MPI_Win_set_errhandler,

RMA_EXT	**External interfaces for One-sided communication (Remote Memory Access)**
	MPI_Win_create_keyval, MPI_Win_delete_attr, MPI_Win_free_keyval, MPI_Win_get_attr, MPI_Win_get_name, MPI_Win_set_attr, MPI_Win_set_name,

RMA_MISC	**Miscellaneous functions for One-sided communication (Remote Memory Access)**
	MPI_Win_c2f, MPI_Win_f2c,

SPAWN	**Process spawning**
	MPI_Close_port, MPI_Comm_accept, MPI_Comm_connect, MPI_Comm_disconnect, MPI_Comm_get_parent, MPI_Comm_join, MPI_Comm_spawn, MPI_Comm_spawn_multiple, MPI_Lookup_name, MPI_Open_port, MPI_Publish_name, MPI_Unpublish_name,

TOPO	**Topology (cartesian and graph) communicators**
	MPI_Cart_coords, MPI_Cart_create, MPI_Cart_get, MPI_Cart_map, MPI_Cart_rank, MPI_Cart_shift, MPI_Cart_sub, MPI_Cartdim_get, MPI_Dims_create, MPI_Dist_graph_create, MPI_Dist_graph_create_adjacent, MPI_Dist_graph_neighbors, MPI_Dist_graph_neighbors_count, MPI_Graph_create, MPI_Graph_get, MPI_Graph_map, MPI_Graph_neighbors, MPI_Graph_neighbors_count, MPI_Graphdims_get, MPI_Topo_test,

TYPE	**Datatypes**
	MPI_Pack, MPI_Pack_external, MPI_Pack_external_size, MPI_Pack_size, MPI_Sizeof, MPI_Type_commit, MPI_Type_contiguous, MPI_Type_create_darray, MPI_Type_create_f90_complex, MPI_Type_create_f90_integer, MPI_Type_create_f90_real, MPI_Type_create_hindexed, MPI_Type_create_hvector, MPI_Type_create_indexed_block, MPI_Type_create_resized, MPI_Type_create_struct, MPI_Type_create_subarray, MPI_Type_dup, MPI_Type_extent, MPI_Type_free, MPI_Type_get_contents, MPI_Type_get_envelope, MPI_Type_get_extent, MPI_Type_get_true_extent, MPI_Type_hindexed, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_lb, MPI_Type_match_size, MPI_Type_size, MPI_Type_struct, MPI_Type_ub, MPI_Type_vector, MPI_Unpack, MPI_Unpack_external,

TYPE_EXT	**External interfaces for datatypes**
	MPI_Type_create_keyval, MPI_Type_delete_attr, MPI_Type_free_keyval, MPI_Type_get_attr, MPI_Type_get_name, MPI_Type_set_attr, MPI_Type_set_name,

TYPE_MISC	**Miscellaneous functions for datatypes**
	MPI_Type_c2f, MPI_Type_f2c,

Bibliography

[1] D. Becker, R. Rabenseifner, F. Wolf, J. Linford: Scalable timestamp synchronization for event traces of message-passing applications. Journal of Parallel Computing **35**(12):595–607, December 2009. 35

[2] D. Becker, R. Rabenseifner, F. Wolf: Implications of non-constant clock drifts for the timestamps of concurrent events. In: *Proc. of the IEEE Cluster Conference (Cluster 2008)*, pp. 59–68, IEEE Computer Society, September 2008. 35

[3] Barcelona Supercomputing Center: Paraver – Obtain Detailed Information from Raw Performance Traces. June 2009. 5, 40

http://www.bsc.es/plantillaA.php?cat_id=485

[4] A. Chan, W. Gropp, E. Lusk: Scalable Log Files for Parallel Program Trace Data — DRAFT. 40

ftp://ftp.mcs.anl.gov/pub/mpi/slog2/slog2-draft.pdf

[5] M. Geimer, F. Wolf, B. J. N. Wylie, B. Mohr: Scalable Parallel Trace-Based Performance Analysis. In *Proc. of the 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, LNCS 4192, pp. 303–312, Springer, Berlin/Heidelberg, September 2006. 1

[6] Gesellschaft für Wissens- und Technologietransfer der TU Dresden mbH: Vampir – Performance Optimization. June 2009. 5, 40

http://vampir.eu

[7] J. Labarta, S. Girona, V. Pillet, T. Cortes, L. Gregoris: DiP: A Parallel Program Development Environment. In *Proc. of the 2nd International Euro-Par Conference*, LNCS 1123, pp. 665–674, Springer, Berlin/Heidelberg, August 1996. 5

[8] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. Version 2.1, June 2008. 3

http://www.mpi-forum.org

[9] W. Nagel, M. Weber, H.-C. Hoppe, K. Solchenbach: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer **12**(1), pp. 69–80, SARA, Amsterdam, January, 1996. 5

[10] Performance Research Lab, University of Oregon: TAU User Guide, chapter *Selectively Profiling an Application*. 27

http://www.cs.uoregon.edu/research/tau/docs/newguide/bk01ch01s03.html

[11] Performance Research Lab, University of Oregon: TAU Reference Guide, chapter *TAU Instrumentation Options*. 27

http://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch01.html

[12] F. Wolf, B. Mohr: Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture **49**(10–11), pp. 421–439, Elsevier, November 2003. 1